DHH Ames

24P.

IN-16839

# Algorithms vs Architectures for Computational Chemistry

*Harry Partridge*
*Charles W. Bauschlicher, Jr.*

January 1986

Research Institute for Advanced Computer Science
NASA Ames Research Center

RIACS TR 86.3



Research Institute for Advanced Computer Science

# Algorithms vs Architectures for Computational Chemistry

Harry Partridge

Research Institute for Advanced Computer Science
NASA Ames Research Center
Moffett Field, Ca. 94035

and

Charles W. Bauschlicher, Jr.

NASA Ames Research Center
Moffett Field, Ca. 94035

## Introduction

In quantum chemistry we determine the properties of atoms and molecules from first principles by solving the time independent Schrodinger equation. The solution algorithm we employ involves a double basis set expansion of the wavefunction $\Psi$ using a variational principle or a perturbation expansion to optimize the parameters[1]. The quantum chemistry techniques are capable of providing accurate atomic and molecular properties such as molecular geometries[2], dissociation energies[3] and transition probabilities[4]. The calculated properties both complement and supplement the available experimental data. In addition, the results can provide qualitative insight of chemical phenomena[5].

The algorithms employed are computationally intensive and, as a

result, increased performance (both algorithmic and architectural) is required to improve accuracy and to treat larger molecular systems. In this work we examine several benchmark quantum chemistry codes on a variety of architectures. While these codes are only a small portion of a typical quantum chemistry library, they illustrate many of the computationally intensive kernels and data manipulation requirements of our applications. Furthermore, understanding the performance of the existing algorithms on present and proposed supercomputers serves as a guide for future program and algorithm development. The algorithms investigated are: a) a sparse symmetric matrix vector product, b) a four index integral transformation and c) the calculation of diatomic two-electron Slater integrals.

In this work we examine the vectorization strategies for these algorithms for both the Cyber 205 and Cray XMP. In addition, we look at multiprocessor implementations of the algorithms on the Cray XMP and on the MIT static data flow machine proposed by Dennis[6].

The Cyber 205 and Cray XMP used in this study are located at NASA Ames Research Center. The Cyber 205 is an 8M word four pipe machine and the Cray XMP is a two processor, 2M word machine with a 16M word SSD (solid state disk). The FTN200 and CFT 1-11 compilers were used for the Cyber 205 and Cray XMP, respectively. For multiprocessing, the CFT 1-13 compiler was used.

The architectures of the Cray XMP[7] and Cyber 205[8] are significantly different so that vectorization strategies differ. From a programmer (or algorithm) point of view the major differences are: Firstly, on the Cyber 205 a vector must have its elements stored contiguously in memory, while on the Cray, successive elements of the vector can be spaced by a constant stride in memory. Secondly, short-vectors do not perform well on the Cyber 205 in comparison with either the scalar or asymptotic vector

performance. On the Cray XMP, vector lengths of 2-3 usually equal scalar performance and the asymptotic rate is approached very quickly. Thirdly, the Cyber 205 has a rather impressive hardware gather/scatter feature that allows random elements to be gathered together to form a vector. On our Cray this is done as a software construct. (The new Cray XMP's have hardware gather/scatter which increases the programming options.) Finally, the Cray can have a solid state disk (SSD) which substantially reduces the IO wait time.

The relatively poor short vector performance and the requirement for contiguous vectors on the Cyber 205 necessitates considerably more care in the algorithm design than is required for the Cray. In general, however, if an algorithm emphasizes long vectors, it will run well on both machines. To some level then, portability considerations suggest that it is best to consider long vector implementations (even if the contiguous memory requirement must be relaxed and gather/scatter used) because this will facilitate running on all vector (and pipelined) machines. There is also evidence indicating that this will facilitate the use of at least some classes of parallel processors[9]. The major disadvantage of this approach for the Cray XMP is that it suggests avoiding some matrix formulations of algorithms. Matrix multiplication on the Cray XMP is very efficient even for fairly small matrices. Similar operations on the 205 are not very efficient unless many matrices can be treated simultaneously.

The computational model in data flow is significantly different from the traditional von Neumann machine[10]. In a data flow machine each instruction (operation) is represented as a directed graph having one or two inputs and an output. Data values move as tokens on the arcs of the graph. An instruction (graph) can execute (fire), consuming its inputs and producing an output token, whenever all of its inputs are present. Since the arcs on the graph represent the data dependencies among the operations, any node with all its inputs present may be fired, regardless of whether other

graphs around it are firing. The architecture has the advantage that it can exploit all of the parallelism inherent in the algorithm and not just that about the program counter, as in a traditional von Neumann machine. The parallelism in the algorithm can thus be exploited regardless of whether or not we have vectors. To facilitate the generation of the data flow graphs, data flow languages have been proposed. These are usually functional languages.

The MIT static data flow machine[6], a specific architecture that has been proposed to implement this model, imposes two additional constraints. The instruction firing rules include the condition that there be no token existing on the output edge, and that the graphs (code) are statically distributed over the available processing elements at compile time. The characteristics of the model architecture studied[11,12] are described in Figure 1. The suggested machine consists of 256 processing elements (PE) interconnected with a cube-type routing network. Each PE has an instruction store and 0.512M words of array memory. Each is capable of between 5 and 8 MFLOPS and each executes the data flow graphs allocated to it. If the result token of each graph is not local to that PE then the routing network is used to transfer the token to other PE's that need the result. The design supports both spatial parallelism (concurrent execution in multiple processing elements) and pipelining (streaming array values from array memory through the PE and functional units). To utilize the machine effectively, it is necessary to keep the floating point units (FP) on each PE busy. If all of the tokens are local to the PE then 10 active instructions are sufficient for the PE to run at the peak FP rate. References to the array memory and network transfers could require that each PE need up to 250 FP active instructions for the pipeline to operate at peak performance. Thus, to obtain near optimum performance on this architecture, it is necessary to maximize local memory (graph) references. The language proposed with this architecture is the functional language VAL[13]. The VAL compiler is being developed and is expected to be able to assist significantly in the transformation. Because the IO system is poorly defined on the

machine at present, we will assume that all implementations of the algorithms will fit in core for this architecture.

From an algorithmic point of view, we will consider the machine to be 256 parallel processors--each with a peak performance of 5 MFLOPS--which are connected with a rather sophisticated network. Our goal is to localize memory references so that each PE can be kept busy (the network transfer rate or IO does not become the rate limiting step). In this way, we can ascertain a rough performance estimate for the data flow machine and also obtain insight into the performance on other non-shared memory multiple processors.

Sparse Matrix Vector Product

The product of a large randomly sparse symmetric matrix times a set of vectors occurs in many applications. In computational chemistry it occurs in constructing the Fock matrix in Self Consistent Field (SCF) calculations[14], in solving linear equations, and in solving for the lowest few eigenvalues and eigenvectors[15]. The scalar FORTRAN for this code is given in Figure 2A. We compute

$$D=HC$$

where H is a symmetric matrix of order n (n= 10000 to 50000) and C and D are of dimension (n,NROW) with NROW $<<$n (typically NROW = 1 to 5). The matrix is assumed to reside on disk and, to reduce IO, the row index of each non-zero element is stored in the low order bits. Initially, the algorithm appears to be essentially scalar because the number of vectors, NROW, is usually too small to obtain effective vectorization--especially for long vector machines--and the matrix is sufficiently sparse (as low as 1%) that the sparsity cannot be ignored. Using the gather/scatter features[16] of the machines, however, one obtains considerable improvement over scalar performance. The non-zero elements of each row of H are stored sequentially. We thus gather together elements of C and D which correspond to the nonzero elements of H,

perform the corresponding vector operation on the compressed C and D and scatter the modified elements of the compressed D back to memory. The code for the Cray is given in Figure 2B; the code for the 205 is similar. The Cray SPAXPY and SPDOT combine the gather/scatter operation with the corresponding floating point operation and avoids a transfer back to memory to form the temporary vector. The timings for the CYBER 205 and CRAY are given in Table 1 in the column labeled VL. The column labeled VS lists the timings for vectorizing over NROW. On the Cray XMP, the VS algorithm is the most efficient algorithm if NROW $\geqslant 7$. On the Cyber 205 and on the Cray XMP, when NROW $\leqslant 6$, the VL algorithm is the most efficient with the Cyber 205 (Cray) vector code being 6.0 (2.2) times faster than the corresponding scalar code. The Cray scalar code is 1.5 times faster than the Cyber 205 scalar code and the vector improvement obtained on the Cray is not as good as on the Cyber 205 since the Cray gather/scatter operations are software operations. Ignoring IO overhead, the Cyber 205 (Cray) performances are 30.8 (and 18.8) MFLOPS in vector mode and at 4.8 (and 7.6) MFLOPS in scalar mode. (The unpacking operation is included in the operation count.) The hardware gather/scatter feature thus yields significant improvement even for codes which appear to be scalar with each operand used only once.

One other important point is that the CPU overhead for FORTRAN IO can be enormous, particularly on the Cyber 205. This is shown in Table 1 under the heading Row IO. The IO can increase the CPU time by a factor of 40! It is important that either large buffer lengths be used or that BUFFERIN or virtual IO be used.

For multiprocessors, the algorithm for the nonsymmetric case has been considered in detail by Reed et. al.[17]. The algorithm for symmetric matrices is complicated slightly by the fact that each element $H_{ij}$ contributes to both $D_i$ and $D_j$. IO and memory concerns still strongly suggest that the symmetric form of H be explicitly utilized. One implementation requires each processor to have

access to all of C with each processor forming a partial result Dp. Each processor reads a record of H and, for these elements, computes the product HC. When all of the records of H have been processed the product D is calculated as

$$D_{ik}=\sum_p Dp_{ik}$$

We have implemented this on the multiprocessor Cray XMP with 3 variations. The code for one of these is given in Figure 2C. Firstly, the file H is split into two separate files and each processor proceeds independently until the final summation. Secondly, each processor works independently but reads from the same file (the system manages locks and unlocks for IO). Finally, one processor manages the IO and then spawns off tasks. All of these approaches yield a total time (summing over the CPU's) which are nearly identical to that for one processor. The efficiency for all 3 methods is similar and is about 99%. In essence, this means that an SCF calculation can be speeded up by nearly a factor of 2 on a two CPU machine by using 1.5 times the memory. (The Fock matrix construction is about 97% of the total SCF time).

On the static data flow machine essentially the same algorithm could be implemented. A sample code for this machine is given in Figure 3 where we have coded the VS algorithm (in VAL) assuming a global memory. The algorithm is easily distributed over the PE's where each PE stores a portion of H in its array memory. The corresponding partial sum Dp is computed by importing the needed elements of the vectors C. Depending upon the size and sparsity of the matrix, each PE will need only part of the elements of C. For the n=20000, 1% nonzero test case, each PE will require about 3/4 of the the elements of C. The rate limiting step of this implementation is therefore the network transfer time to transmit C and Dp. For the above example the IO time would be about 0.16 seconds while the total CPU time would be only 0.01 sec. The total execution time is thus 0.17 seconds.

Another implementation has each PE store about m=n/256

elements of C and of D. The matrix is then divided (sorted) into $M^*(M+1)/2$ subblocks each spanning approximately an equal number of rows and columns (m) with each subblock allocated to a PE (M=22 square subblock would allocate one block to 253 of the PE's). Each PE would then require no more than 2m elements of C and would calculate no more than 2m elements of Dp. Furthermore, each element of D could be summed requiring no more than M elements of Dp. Thus, the number of words to be transferred over the network for each PE is m(M+2). For our sample problem, m=79 and M=22; the network transfer time is less than 0.01 seconds.

Four Index Transformation

The four index transformation is needed to transform the two electron integral file (a function of four variables, F(i,j,k,l), with respect to a different basis set.

$$G(p,q,r,s)=\sum_{i,j,k,l} C_{p,i} C_{q,j} C_{r,k} C_{l,s} F(i,j,k,l)$$

The algorithm[18] involves the formation of partial sums to reduce the computational complexity but it requires a significant shuffling (reordering of the partially transformed integrals) half-way through the calculation to keep the memory references local. To obtain efficient vectorization it is necessary to treat molecular symmetry explicitly, which essentially blocks the function F into relatively dense subunits.

If we define the $n_i$ by $n_j$ matrices $F^{kl}$ as the corresponding subunits of F then the first half transform may be expressed as a sequence of similarity transforms, $H^{kl}=C^T F^{kl} C$. If we shuffle the elements of H to form $H'$ with $H'^{ij}_{kl}=H^{kl}_{ij}$, then G may be formed as another sequence of similarity transforms. The algorithm is thus broken into the following steps:

1) The expand step to form the matrices, $F^{kl}$. Only the elements of F greater than some threshold are usually stored on disk. In addition, for many of the symmetry blocks of F there are

restrictions on the range of the indices since only the unique integrals are stored.

2) First half transform, denoted as MXM1.

3) Sort or shuffle step to reorder the partially transformed integrals. Since the integral file does not fit into memory, random access is used to perform a bin sort.

4) Second half transform, denoted as MXM2.

The Cray timings for a typical case are given in Table 2. It is important to note that the scalar steps of the order $n^2$, $O(n^2)$, becomes very important on a vector machine even though the vector operations are $O(n^3)$. Careful consideration must be given to these sections even though on a scalar machine the improvement will be small. The Cray matrix multiplication routines are very efficient for performing the similarity transforms even for matrices as small as 4 by 4 (see Table 3). On the Cyber 205, the vector lengths for a matrix multiply are too short to provide efficient vectorization. Since we have $O(n^2)$ similarity transforms to do, however, we can perform many of these simultaneously thereby obtaining vector lengths of $O(n^3)$. The timings for these transformations are given in Table 4 where NM is the number of transformations done together. The periodic gather/scatter feature of the 205 is used to reorder the integrals to obtain contiguous vectors. Even with the gather/scatter overhead, the half-transformation computation rate is 300 MFLOPS for this example.

The algorithm transfers trivially to a multiprocessor environment because each of the $O(n^2)$ similarity transforms can be performed independently. The reshuffling on the multiprocessor should cause no difficulty since each processor can perform independent bucket sorts with only the need to synchronize before starting the second half transform. On the static data flow machine, we will again assume the integral file will fit in memory. For the test case in Table 2 the CPU time will be on the order of 0.12 sec and the shuffle time on the order of 0.01 sec.

Diatomic Slater 2-electron Integrals

The numerical evaluation of $O(n^4)$, where n=100-300, diatomic exponential (Slater) type orbital two-electron integrals is one of the most computationally intensive steps in our codes. The algorithm[20] uses the Neumann expansion for $r_{ij}^{-1}$ and each term in the expansion involves an iterated double numerical integration. The integrals are required to have a high degree of (absolute) accuracy--typically $<10^{-10}$--to avoid numerical linear dependency problems. A charge distribution approach is implemented[20]. For each term in the expansion the set of $n^2$ charge distributions is calculated and all possible vector dot products (length O(500)) are formed. Given sufficient memory to hold all of the charge distribution quantities the CPU time is dominated by the dot products.

The code scans the list of integrals, allocating memory and initializing each charge distribution (CD) it encounters. When memory is exhausted, it computes the scanned integrals. This process is repeated until all of the integrals have been computed. The algorithm performance improves with increased memory since considerably fewer CD quantities are calculated (see Table 5). The code again demonstrates the importance of scalar sections. In scalar mode about 95% of the CPU time is spent performing dot products compared to only 35% in vector mode.

There are many organizations possible for implementing this algorithm on multiprocessors. Since any number of the $O(n^4)$ integrals may be computed independently, the simplest approach is to partition the integral list and have each processor work on separate partitions. If we define the speedup in performance for n processors to be the ratio of the performance of n processors to that of one processor, then the speedup for this implementation is n since each of the partitions is independent. The implementation could be rather inefficient, however, since most of the CD quantities will need to be recomputed many times. On some architectures this might be the optimal implementation anyway. This might occur, for example, if

there were $O(n^4)$ processors or if the interconnection network were sufficiently slow (the definition of sufficiently slow would depend on the amount of memory available per processor). On neither the Cray XMP nor the MIT static data flow architecture is this algorithm optimal. On the Cray XMP the CPU's share memory so that twice the memory is available. Thus, we can store twice the number of CD quantities and compute many more integrals before needing to reinitialize. In effect, we will be computing considerably fewer CD quantities so that our efficiency would be greater than 2. On the static data flow architecture, we can divide the $O(n^2)$ CD quantities among the PE's (m per PE) and partition the integral file into subblocks. Each processor would need at most 2m CD quantities, and each would compute $O(m^2)$ integrals. Also, each PE would need to only import the CD quantity itself and not the associated tables needed to form the CD quantity. The algorithm is thus expected to perform very well, close to the 1.2GFLOPS limit, without requiring considerable redundant calculations.

In conclusion, computational chemistry codes can perform well on both the Cyber 205 and the Cray XMP. Algorithm development and coding considerations are more involved for the Cyber 205 but impressive computational chemistry packages have been developed for both the Cyber 205 [21] and the Cray XMP[22]. In addition, there is a considerable degree of parallelism in the algorithms that can be easily exploited. Considerable algorithmic development will be required for some steps (notably the multiconfiguration self consistent (MCSCF) and configuration interaction (CI) steps) to reduce the network traffic, particularly on non-shared memory architectures. The improved performance which will be obtained from multiprocessors will significantly extend the systems that can be studied.

The authors would like to acknowledge helpful discussions with G. B. Adams and M. L. Patrick and to thank J. Dennis, W. Ackerman, and G. Guang-Rong for the help during the data flow workshop.

**Bibliography**


1. H. F. Schaefer, "The Electronic Structure of Atoms and Molecules", Addison-Wesley, Reading,Mass. (1972).
2. B. H. Lengsfield, A. D. McLean, M. Yoshimine, and B. Liu, 79, 1891 (1983).
3. C. W. Bauschlicher, B. H. Lengsfield and B. Liu, J. Chem Phy., 77, 4084 (1982); S. R. Langhoff, C. W. Bauschlicher, and H. Partridge, "Theoretical Dissociation Energies for Ionic Molecules", in "*Comparison of ab initio Quantum Chemistry with Experiment*", ed. R. Bartlett, D. Reidel, Boston, Mass (1985).
4. M. Larsson and P. E. M. Siegbahn, J. Chem. Phys., 79, 2270 (1983).
5. M. Seel and P. S. Bagus, Phys. Rev. B., 28, 2023 (1983).
6. J. B. Dennis, "Data Flow Ideas for Supercomputers," Proceedings of the IEEE COMPCON, p15, February 1984.
7. Fortran (CFT) Reference Manual SR-0009, Cray Research Inc, Mendota Heights, Minn, 1984.
8. Fortran 200, Version 1, Control Data Corp., Mineapolis, Minn, 1984.
9. D. B. Gannon and J. Van Rosendale, IEEE Tran. Comp., 33, 1180 (1984).
10. A. L. Davis, R. M. Keller, "Data Flow Program Graphs," Computer. 15, 26 (1982).
11. The data flow workshop was conducted by RIACS in September 1984 to assess the effectiveness of data flow programming using a specific data flow machine for computational problems of interest to NASA and DARPA. The workshop was conducted by three researchers from MIT: J. Dennis, W. Ackerman, and G. Guang-Rong.
12. G. B. Adams III, R. L. Brown, R. J. Denning, "Report on an Evaluation Study of Data Flow Computation", in preparation.

13. W. B. Ackerman, J. B. Dennis, "VAL--A Value-oriented Algorithmic Language: Preliminary Reference Manual," MIT/LCS/TR-218, Massachusetts Institute for Technology, Cambridge, MA (Jun 1979); W. B. Ackerman, Computer, 15, 15 (1982).

14. R. C. Raffenetti, Chem. Phys. Lett., 20, 335 (1973).

15. E. R. Davidson, J. Comp. Phys., 17, 87 (1975); B. Liu, NRCC report on the workshop "Numerical Algorithms in Chemistry: Algebraic Methods", Lawrence Berkeley Laboratory, Berkeley, Ca, p49, 1978.

16. For example in FORTRAN the gather is

```
      DO 10 I=1,N
10    A(I)=B(INDX(I))
```

17. D. A. Reed and M. L. Patrick, "Iterative Solution of Large, Sparse, Linear Systems on a Static Data Flow Architecture: Performance Studies", in preparation; D. A. Reed and M. L. Patrick, "Parallel, Iterative Solutions of Sparse Linear Systems: Models and Architectures," Parallel Computing, to appear.

18. M. Yoshimine, IBM Report R. J. 555, 1969, San Jose, Ca.

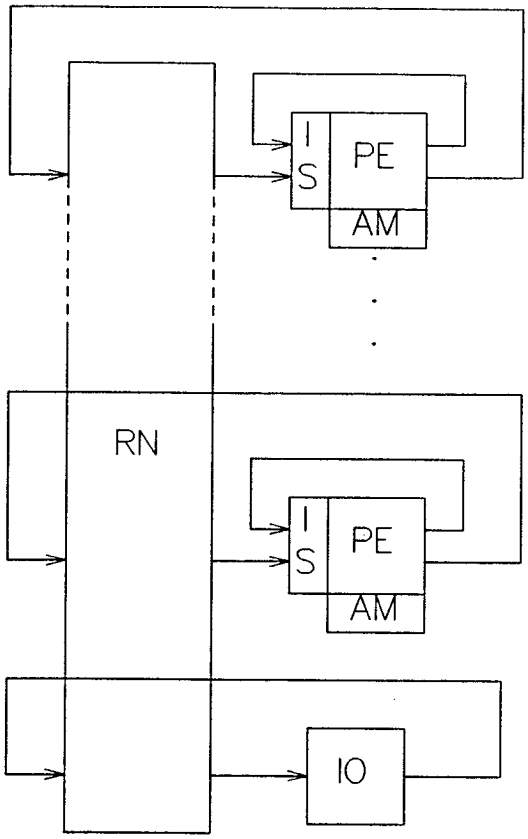
19. K. Rudenberg, J. Chem. Phys., 19, 1459 (1951).

20. S. Hagstrom, QCPE 10, 252 (1975); S. Hagstrom and H. Partridge (unpublished).

21. R. Ahlrics, H. J. Bohm, C. Ehrhardt, P. Scharf, H. Schiffer, H.Lischka, and M. Schindler, J. Comp. Chem. in press.

22. V. R. Saunders and J. H. van Lenthe, Mol. Phys., 48, 923 (1983).

Figure 1: Static Data Flow Machine Architecture[a]




RN Routing Network. 512 by 512, 16 bit data paths, operates at > 5MHz, average rate of transmitting FP packets 0.25 MHz from a single PE to another.

PE Processing Elements. 5 to 8 MFLOPS with two 1.25 to 2 MFLOP multipliers. 256 PE's in the system.

IS Instruction Store. 1024 cells for FP instructions, 1024 for others.

AM Array Memory. Size not fully determined. At least 256K 64 bit words per PE.

IO Input Output. Includes mass memory, host processor, and display systems. 256 paths through the RN are reserved for IO.

[a]This is figure 2 for [12]

**Figure 2: FORTRAN code for sparse matrix vector product**

```
C     Row IO VERSION of D=HC.
C     NROW IS THE NUMBER OF VECTORS
C     N IS MATRIX DIMENSION
C     IU IS DISK UNIT
```

**A: Scalar Code**

```
      DO 1 I=1,N
      READ(IU)NZ,(BUF(J),J=1,NZ)
      DO 2 K=1,NROW
      DO 3 JX=1,NZ
      J=AND(BUF(JX),MASK16)
      HC(J,K)=HC(J,K)+C(I,K)*BUF(JX)
      HC(I,K)=HC(I,K)+C(J,K)*BUF(JX)
3     CONTINUE
2     CONTINUE
1     CONTINUE
```

**B: Cray XMP Code**

```
      DO 1 I=1,N
      READ(IU)NZ,(BUF(J),J=1,NZ)
      DO 3 J=1,NZ
      ISC(J)=AND(BUF(J),MASK16)
3     CONTINUE
      DO 2 K=1,NROW
      HC(I,K)=HC(I,K)+SPDOT(NZ,C(1,K),ISC,BUF)
      CALL SPAXPY(NZ,C(I,K),BUF,HC(1,K),ISC)
2     CONTINUE
1     CONTINUE
```

C: **Cray XMP multiprocessor code.**

```
      SUBROUTINE D1(C,HC,NROW,N,OUT,OUT2,HC2)
      COMMON /UNITS/IHU
      DIMENSION OUT(10240),OUT2(10240),ISC(20000),ISC2(20000)
      DIMENSION C(N,NROW),HC(N,NROW),HC2(N,NROW)
      DIMENSION IDTASK(3)
      EXTERNAL HCM
      IDTASK(1)=3
      IDTASK(3)=7777
      REWIND IHU
  101  CONTINUE
      READ(IHU,END=88,ERR=88)OUT
      CALL TSKSTART(IDTASK,HCM,C,HC,NROW,N,OUT,ISC)
      READ(IHU,END=88,ERR=88)OUT2
      CALL HCM(C,HC2,NROW,N,OUT2,ISC2)
      CALL TSKWAIT(IDTASK)
      GO TO 101
   88  CONTINUE
      CALL TSKWAIT(IDTASK)
      RETURN
      END
      SUBROUTINE HCM(C,HC,NROW,N,OUT,ISC)
      DIMENSION C(N,NROW),HC(N,NROW),OUT(1),ISC(1)
      NN=AND(OUT(1),.NOT.MASK(30))
      MMM=.NOT.MASK(48)
      IU=1
      IXX=0
    1  CONTINUE
      IU=IU+1
      IXX=IXX+1
      IF(IXX.GT.NN)RETURN
      NUM=OUT(IU).AND. .NOT.MASK(34)
      I=SHIFTR(OUT(IU),30)
      DO 2 JX=1,NUM
      ISC(JX)=AND(OUT(JX+IU),MMM)
```

```
2    CONTINUE
     DO 4 K=1,NROW
     HC(I,K)=HC(I,K)+SPDOT(NUM,C(1,K),ISC,OUT(IU+1))
     CALL SPAXPY(NUM,C(I,K),OUT(IU+1),HC(1,K),ISC)
4    CONTINUE
     IU=IU+NUM
     IF(I.LT.N)GO TO 1
     RETURN
     END
```

**Figure 3: VAL code for sparse matrix vector product**

```
function hcxsp(
% function to multiply d=hc where h is a randomly sparce matrix and
%        c and d are (a set of) vectors. Assumption are that both
%        c and d fit in "memory" (are randomly accessible). h is
%        a real symmetric matrix of dimension n with only the nonzero
%        elements stored (only the lower traingular elements of h are
%        stored). The number of nonzero elements of h can be large.
%
%
      n,ncols:integer;              % dimension of matrix, number of
%                                      vectors c (and d)
      c:array[array[real]];
      argh:array[array[real]];       % non zero elements of h
      argindex:array[array[integer]]  % for a given row the column index
  returns
      array[array[real]]             % return d
          )
 for d:=array_fill(1,ncols,array_fill(1,n,0.));
    i:=1;
%
    do if i >n then d
    else
       let
        h:=argh[i];
        index:=argindex[i];
        nok:=array_size(h);

      in iter d:=
       forall kcol in [1,ncols]
        construct
         for k:=1;
            col:=d[kcol]
         do
          if k>nok then col
```

```
            else let
             x:=col[i:col[i]+h[k]*c[index[k],kcol]];
             y:=if k=nok then x
                else x[index[k]:col[index[k]]+h[k]*c[i,kcol]]
             endif;
            in
             iter col:=y;
                  k:=k+1 enditer
             endlet
            endif
          endfor
        endall;
        i:=i+1
       enditer
      endlet
     endif
   endfor
endfun
```

**Table 1. Summary of CPU times (in seconds) for the Sparse Matrix Vector Product[a].**

| | Cray XMP | | | | Cyber 205 | | | |
|---|---|---|---|---|---|---|---|---|
| NROW | 1 | 3 | 5 | 10 | 1 | 3 | 5 | 10 |
| S[b] | 1.59 | 4.70 | 7.83 | 15.63 | 2.42 | 7.23 | 12.04 | 24.04 |
| VL[c] | 0.74 | 2.00 | 3.28 | 6.48 | 0.44 | 1.22 | 1.99 | 3.94 |
| VS[d] | 4.00 | 4.10 | 4.19 | 4.34 | 9.42 | 9.45 | 9.53 | 9.63 |
| Row IO[e] | 1.89 | 3.17 | 4.45 | 7.56 | 18.52 | 20.53 | 22.76 | 29.53 |

[a]The test case is for a matrix of order 20000 and 1% sparse.

[b]Scalar timings.

[c]Vectorization along row. The Cray code is given in Figure 2b.

[d]Vectorization over number of vectors, NROW.

[e]Each column of H is read as a separate binary read: Read(file)N,(BUF(I),I=1,N) where N is the number of nonzero elements in this row.

**Table 2. Four Index Transformation timings for the Cray (CPU seconds)**

$C_{2v}$ Symmetry

| | | | | |
|---|---|---|---|---|
| AO's | 48 | 26 | 26 | 11 |
| MO's | 45 | 24 | 24 | 11 |

| | CDC 7600 | FORTRAN | MXM | MXM[a] | Operations[b] |
|---|---|---|---|---|---|
| Expand | -- | -- | 1 | 1 | $n^2$ |
| MXM1 | -- | -- | 32 | 24 | $n^3$ |
| Sort | 50 | 37 | 37 | 14 | $n^2$ |
| MXM2 | -- | -- | 6 | 6 | $n^3$ |
| Total | 700 | 126 | 76 | 45 | |

Symmetry Block ($A_2A_1$ | $B_2B_1$)

| | | | | |
|---|---|---|---|---|
| AO's | 58 | 30 | 30 | 13 |
| MO's | 58 | 30 | 30 | 13 |

| | Scalar | FORTRAN[c] | MXM[a] |
|---|---|---|---|
| Expand | 0.32 | 0.01 | 0.01 |
| MXM1 | 19.81 | 1.93 | 0.54 |
| Sort | 2.35 | 1.28 | 1.28 |
| MXM2 | 20.40 | 4.50 | 0.48 |
| Total | 42.13 | 7.88 | 2.50 |

[a]For this case the scalar code has been improved and $C^T$ stored to minimize memory bank conflicts.

[c]Operations for each simularity transform, yields an overall $n^5$ algorithm.

[c]With default vectorization.

**Table 3. Matrix multiplication timings (in sec) for the Cray XMP and Cyber 205[a]**

| | Cray XMP | | Cyber 205 | |
|---|---|---|---|---|
| n | MXM | FORTRAN | FORTRAN | Scalar |
| 4 | 0.0000056 | 0.0000354 | 0.000069 | 0.000062 |
| 8 | 0.0000145 | 0.000121 | 0.000253 | 0.00037 |
| 16 | 0.0000603 | 0.000484 | 0.000693 | 0.00269 |
| 32 | 0.000378 | 0.00204 | 0.00282 | 0.0206 |
| 64 | 0.00273 | 0.01002 | 0.0120 | 0.161 |
| 128 | 0.0216 | 0.0558 | 0.0539 | 1.274 |
| 256 | 0.171 | 0.359 | 0.257 | 10.389 |

[a]These timings are the average of 10 executions in batch mode.

**Table 4. Cyber 205 timings (in seconds) to perform the half-transform of the four index transformation**

| NM[a] | MXM1[b] | MXM2[b] | MXM[c] |
|---|---|---|---|
| 1 | 10.10 | 4.49 | 116.2 |
| 5 | 2.26 | 1.11 | 84.3 |
| 10 | 1.28 | 0.69 | 23.1 |
| 20 | 0.79 | 0.48 | 17.9 |
| 50 | 0.50 | 0.35 | 14.8 |
| 100 | 0.40 | 0.307 | 13.8 |
| 400 | | | 13.3 |
| 754 | | 0.271 | |
| 900 | 0.31 | | |
| Cray | 0.54 | 0.48 | |

[a]NM is the number of similarity transforms performed simultaneously.

[b]Timings for the symmetry block $(A_2A_1 \mid B_2B_1)$ of table 2.

[c]Timings for square case with the number of AO's=64.